Racing with Deep Reinforecment Learning

Walter A. Freire, Kevin M. Freire
andrei.freire@ryerson.ca, kfreirea@ryerson.ca
Ryerson University, Toronto, Canada.

***Abstract*—The group members are Walter Freire and Kevin Freire and the project topic we will be working on is autonomous driving using deep reinforcement learning. The project uses the DeepRacer platform for simulation and training of a model. This article focuses on implementing a custom neural network architecture, with custom reward functions based on the parameters provided by DeepRacer.**

I. Intorduction

Reinforcement Learning (RL) has been used to accomplish various robotic tasks such as manipulation, navigation, flight, interaction, motion planning and many more. When implementing an RL model into the real world, due to its high sample complexity, safety requirements and cost it is common to train the RL agent in a simulation. In Autonomous driving, RL require many disciplines and equipment such as access to a physical robot/car with proper sensors, an accurate robot model for simulation to avoid damaging expensive hardware in initial development, a diverse training mechanism and custom modification of the training procedure such as modifying the neural network architecture and the loss function [1]. For our experiment we used Deepracer as a guideline to implement and modify an existing jupyter notebook using AWS Sagemaker and Robomaker APIs.

We decided to get started with Deepracer because it supports state-of-the-art deep RL algorithms [29], simulations with OpenAI Gym interface [17], distributed rollouts and integration with cloud services. DeepRacer uses a 1/18th scale car of a physical robot that uses RL for navigation of a race track with a fisheye lens camera [1] with other specs such as GPU, live streaming view and more. In this paper the robot model in DeepRacers simulation is used along with the multiple provided race tracks. With the provided sources the RL model policy is trained with different simulation parameters and multiple tracks in parallel using distributed rollouts.

DeepRacer is a great platform for learning an end-to-end policy for navigating through a race track, that uses a single grayscale camera image as observation and discretized throttle/steering ad actions. The training is done in simulation using Proximal Policy Optimization (PPO) algorithm [31], which can converge in under 5 minutes and about 5000 simulation steps. Deepracer provides an interactive experience in training a vehicle to drive itself through a race track with minimal supervision. We were mainly interested in the available jail-broken version in order to modify their source code to implement our own network architecture, customize the action-space, the reward functions and much more. This is because Deepracer was very limited to what we can do and were restricted to their built in architectures. For this paper we implemented a simulation using AWS Robomaker and Training Deep reinforcement algorithm using The Sagemaker API to build a custom deep architecture and custom reward function in a built simulation environment built by AWS.

The purpose of selecting this project was to learn and understand the effect of training a vehicle with RL using deep neural networks with large datasets of videos that require high computational power using the Clipped PPO algorithm and customized reward function. [From result based on kevin paper] States that deeper networks reduce training time in self driving vehicles. This is an observation we will attempt to capture in our project. The rest of the report consists of the problem statement along with the environment we used, the custom deep architecture and reward function, the reinforcement algorithm, experiment setup , results, discussion of results. Finally we will be showing code snippets in our appendix with custom code but the full implementation will be available on Github.

II. Problem Statement and Environement/Dataset

The main objective of this problem is autonomous racing. This problem requires the agent (car) to stay in the track and drive itself while completing the track as fast as possible. This problem is divided into separate sections which include:

1) Develop the deep neural network architecture
2) Creating the simulation environment.
3) Create the reward function and action space.
4) Create the proper hyper parameters for training.
5) Collect results, test and compare to AWS shallow network.

To develop a neural network we realize that there are many implementations to this but we decided that we wanted to learn and observe the effects of a deep neural architecture consisting of 8 convolutional layers and 2 dense layers with each layer going through a RELU activation function. To create our simulation environment we used AWS Sagemaker and Robomaker along with a AWS Simulation package to have a simulation environment out of the box which made this project possible in the short time frame. AWS provided us with a simulation environment of many racetracks which we decided to perform our experiment on the racetrack as seen in Figure 1.

In order to train our agents one of the most important steps was our reward function and action space. Our action space was a steering angle and speed of the agents. Specifically the steering angle was from a range between -30 $^\circ$ to +30 $^\circ$ and the speed from 0.5 to 2 *m/s*. Along with one sensor the front facing camera. We did this for a continuous action space to

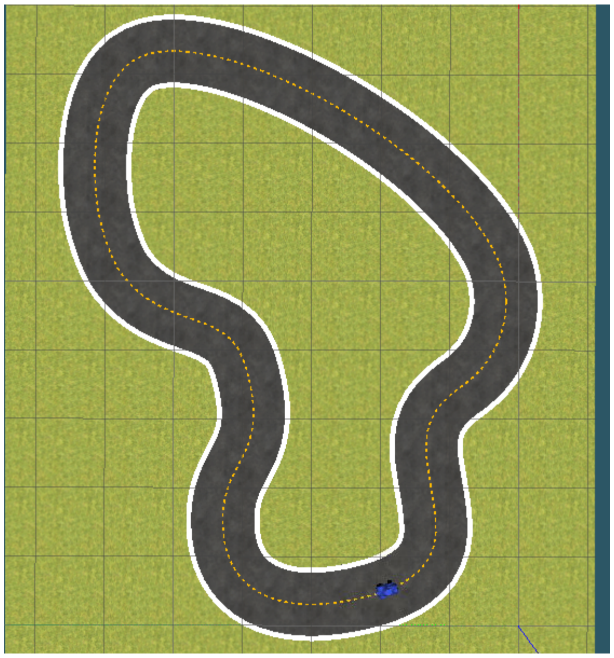

Fig. 1. A Simplified Environment

implement and train using the Clipped PPO algorithm. Finally our rewards function was very complex as we returned specific rewards depending on the state of the agent. For example the agent would half a wheel of track it would return a negative reward, or if the steering angle was not correct it would return negative reward. If the agent had desirable actions at a specific state we would return a positive reward. The parameters we worked with for the reward function are:

1) Flag to indicate if the agent is on the track
2) Distance in meters from the track center
3) Flag to indicate whether the agent has crashed.
4) Percentage of track completed
5) Flag to indicate whether the agent has gone off track.
6) The width of the track
7) The agents' speed in meters per second (m/s)
8) The number steps completed
9) The agents' steering angle in degrees
10) Flag to determine if an agent is left of the center line

The track defines where the agent can go and what state it can be in, thus explores the environment to collect data in order to train the neural network. The agent's data is a single grayscale monocular image captured by the front-facing camera as input which represents the agent's current state. Depending on the current state an action from the agent corresponds to a move at a particular speed and steering angle. The reward is the score given as feedback to the agent when the action at a given state is taken. The reward is returned by a reward function where you can define a reward function to specify what is a desirable or undesirable action.

In the simulator the agent explores the environment and builds up experience, the experiences collected are used to update the neural network periodically and the updated model is used to generate more experiences. A simplified example of the environment is shown in Figure 2. By observing Figure 2, each state represents an individual state. The vehicle is allowed to move up or down while facing in the direction (left to right). In order to achieve the shortest path it must follow a straight line, thus achieving the highest possible reward. However, if the vehicle steers off track it will not receive any reward or receive a negative reward(depending on reward function).

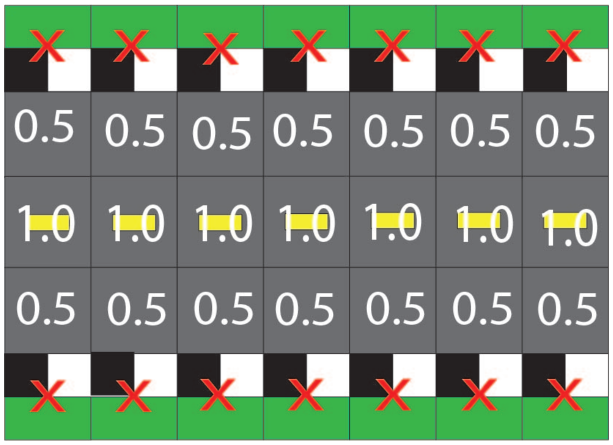

Fig. 2. A Simplified Environment

The next part was to identify the proper hyper parameters. We felt it was easier to keep some default hyper parameters since we implemented our own architecture and reward functions and we did not want to fine tune the models as much because training would cost us a lot of money and doing a fine tuning test would be too costly for this project. Finally in the last section we needed to record the correct results and since this was a race we looked at time completion and how many times the agent went off track during the test. For the training however we collected the reward it obtained during each episode.

III. Methods and Model

A. The Architecture

To implement our architecture we chose to go with a really deep implementation which consists of 8 convolutional layers and 2 denslayears with RELU activation functions after each layer. The architecture is shown in Figure 3, for the convolutional design we picked these many layers mainly for educational purposes to observe the results in having this many convolutional layers. For our final test we will be running a shallow network to see how it compares to our deep architecture. The shallow network architecture provided by AWS is shown in the appendix. Finally the Dense layers have a dropout rate of 50% to handle overfitting. This way the deep architecture does not overfit on one race track and we can see how it trains on other tracks.

B. Action Space

As mentioned above we used an action space such that the agents can perform an action that consists of a steering angle and a speed. Since this is a continuous action space we have many possibilities. The actions that the agent can take is between -30 ° to +30 ° and between 0.5 to 2 *m/s*.

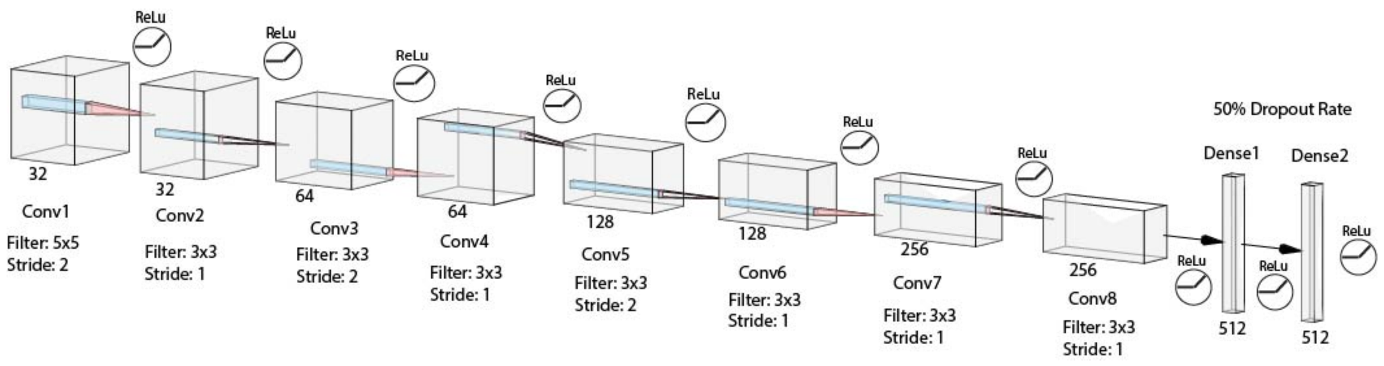

Fig. 3. A Simplified Environment

The angles were chosen based on the maximum limits that amazon provides, the speed we specifically chose to be 2 *m/s* to be our max instead of 4 *m/s* because we noticed that starting of training at really fast speeds can delay the training a lot, specifically when we are just starting to train. A question that arises is whether we can slowly transfer and learn from models that were trained at slower speed and fine tune it changing the action space and train it at higher speeds.

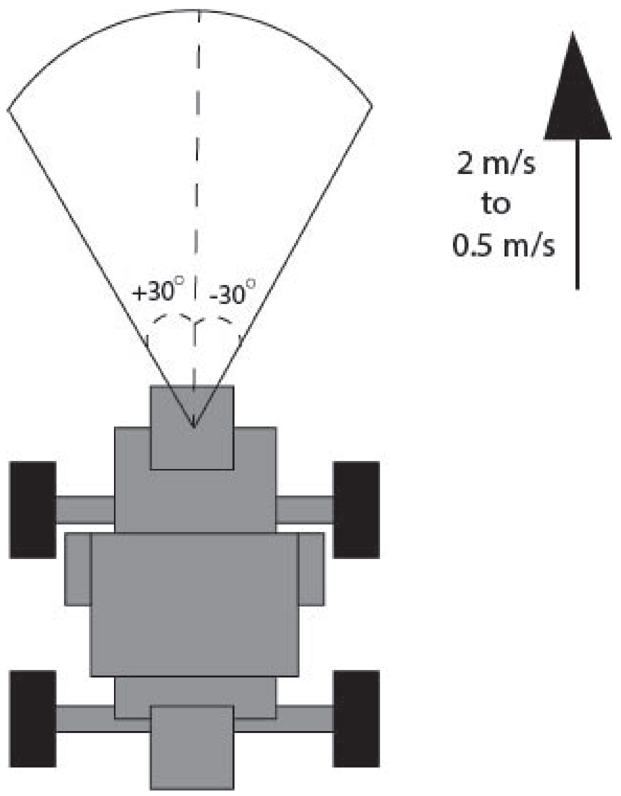

Fig. 4. Simplified car model with max steering angles and speed range

C. Reward Fcuntion

The reward function that we used for all the tests are based on several considerations that affect what the returned reward. The reward function that we used is in the appendix and below is a summary of how we implemented it to train our agent:

1) If crashed or off track return -1 reward (handles terminal state)
2) If car is on the left lane and all the wheels are not on the track and steering angle is to the left return -1 reward (handles terminal state because the next action will be terminal state since it will drive off track)
3) If car is on the right lane and all the wheels are not on the track and steering angle is to the right return -1 reward (handles terminal state because the next action is terminal state since it will drive off track)
4) Handle the opposite reward for steps b and c, return a reward based on the steering handle that return them back to the track (handles desirable action)
5) If no wheels are off track and agents are in a desirable state return a +1 reward and if the speed is fast proportionally add to reward. (handles desirable state)
6) Finally return a reward of +10 if the agent finished the track in 100 steps or less (handles a positive terminal state)

D. Reinforcement Algorithm

Proximal Policy optimization (PPO) is currently a standard use in self driving vehicles. AWS provided two algorithms: the Soft Actor Critic (SAC) Algorithm and Clipped PPO algorithm. We decided to go with CPPO because we wanted to use an algorithm that's popular in self driving vehicles, and for its on-policy learning algorithm which is done by PPO. On-policy algorithms tend to be more stable and exploit exploration which is why we decided to train it using the Clipped PPO algorithm. We did not do any modification to the algorithm but rather just used the API which was implemented by intel RL Coach framework used within AWS Sagemaker.

E. Hyper Parameters

As mentioned above we did not want to focus on fine tuning the model because it was too costly using AWS resources. We focus on two sets of hyper parameters for our tests in order to observe which is better. Shown in Table I are the hyper parameters used for the experiments.

F. Testing Metrics

Finally for the training metrics we wanted to observe some important information. Since this is a race we observe 3

TABLE I
HYPER PARAMETERS USED FOR TRAINING

CPPO Hyper Parameters	**Set 1**	**Set 2**
Batch size	64	64
Epochs	10	15
Learning rate	0.0003	0.0025
Exploring type	Categorical	Categorical
ϵ-greedy	1	0.9
Beta entropy	0.01	0.01
Discount factor	0.95	0.95
Loss type	Huber	Huber
# of episode between training	20	20

metrics. The first was the completion of the track, this was important because it must complete the track fully to finish the race. Second we wanted to record how often the agent went off track and finally the most important result is to see how fast the agent completed the track. Track completion, off track count and time to complete the track were the metrics we used to compare the models we trained.

G. Model Comparison

Since we wanted to test the effect of training a deep network using reinforcement learning we needed to test and compare against another architecture. AWS provided a built in architecture which we used to start our test that used a shallow architecture which only consists of 3 convolutional layers and no dense layers. The convolutional layers where the first layer has 32 filters using 8×8 filter and stride of 4 followed by a ReLU activation function. The second layer has 64 filters using 4×4 fitler and stride 2 followed by a ReLU function and finally the third layer has 64 filters again using 3×3 filter and stride of 1. We ran the same test we ran in our architecture in order to compare results fairly. As mentioned earlier, the object is to observe if a deeper network can reduce training time in self driving vehicles.

IV. Results and Discussion

V. Implementation and Code

As mentioned throughout the paper we used Amazons's Sagemaker API and python's python development environment. Sagemaker Library used the RL Coach from Intel for the reinforcement learning Algorithms. The code is available on github along with our modification of the code we implemented. The github repo is *https://github.com/wfrei020/DeepRacer-Freire*, please read the README file. For the simulation, amazon used robomaker and we implemented it and ran using a GPU. The original notebook provided by AWS for Deepracer was outdated and we spent a lot of effort and time upgrading it to the newest version. The notebook was available at https://github.com/aws/amazon-sagemaker-examples and we upgraded it using the help of issue number 2055. This helped us upgrade a lot of the packages. Finally we also had to make the notebook compatible for CPPO. THe notebook that AWS provided only worked for the SAC algorithm. We really wanted to work with CPPO so we spent several days debugging and adding some code fix to allow the use of CPPO. In the Appendix we have some important code we implemented for our project but we will not be including all the fixes throughout the API as it is too much and not important to the project. Finally in general we used AWS Dockers, Robomaker, Sagemaker ML libraries, and S3 bucket for storing our models.

VI. References

Please make sure you cite any sources you have used to complete the project

[1] DeepRacer Article [29] Deep racer reference 29 [17] Deep racer reference 17 [31] Deep racer reference 31